Mini Project 2 - Generators

...

By Aaron Dixon, Emanuel Jose, and Thomas Mendez

Language Specification

Overview:

- Language is based on:
  - Arithmetic expressions
  - Variable and mutations
  - Sequential execution
- New constructs introduced
  - Generators
  - Yield
  - resume
- Enables pausable execution flow

Language Specification cont.

Lexical Conventions

- Programs are written as S-expression using either () or {}. Both forms are equivalent.
- Numbers
  - Any valid racket numeric for example 0, 2, 3.14, or -7
- Identifiers
  - Names used for variables and generators
  - Can be any valid Racket symbol except reserves words for example x, my-gen, counter

Language Specification cont.

Grammar <expr> ::=

- <number>
- | <id>
- | {+ <expr> <expr>}
- | {let1 {<id> <expr>} <expr>}
- | {begin <expr> ...}
- | {set! <id> <expr>}
- | {generator <expr> ...}
- | {yield <expr>}
- | {resume <expr>}

The expression can be a number or identifier representing literal values. The language supports core computation features like addition, and variable binding which introduces a new variable within the scope. It also incorporates sequencing with begin allowing multiple expressions to run in order, and mutation with set! Which updates existing variables.

The generator enables control over the program execution. The generator expression creates a generator object without immediate execution and inside the generator, the yield expression allows the program to pause the execution and return a value saving its current state. The resume expression can restart the suspended computation form where it left off.

Language Specification cont.

Core language constructs

- Numeric literal <number> : evaluates a numeric value
- Variable Reference <id>: looks up the value currently bound to <id> in the env
- Addition { + <expr> <expr> }: evaluates both sub-expressions (left then right) and returns the sum
- Local binding { let {<id> <expr>} <expr>}: evaluates the expression, binds its value to <id>, then evaluates the body expression in that extended scope
- Sequencing {begin <expr> <expr>...} : returns the value form the last expression (left to right)
- Mutation {set! <id> <expr>} : evaluates <expr> and updates the binding of <id> in the current env to the resulting value.

Language Specification cont.

Generator constructs

- Generator {generator <expr> ...} : creates and returns a generator value without executing the body. A generator value is a stateful object that tracks whether it is fresh (never been resumed), suspended (paused at a yield with saved continuation) or exhausted (body has completed and no more values can be produced.

- Yielding a value {yield <expr>} : this only appears inside the body of a generator. Evaluates <expr> to produce the yielded value then, suspends the generator then returns the yielded value to the caller of the most recent resume.

- Resuming a generator {resume <expr> } : takes a generator and continues its execution. If the generator is fresh, it starts for the beginning. If suspended, it continues from the last yield. If exhausted, it raises an error.

Evaluation order

- All constructs evaluate sub-expressions left to right
  - Both operands of + are evaluated left to right before adding
  - In begin, expressions are evaluated strictly in sequence
  - In resume, the generator expression is fully evaluated before resumption

Implementation

As a form of lazy evaluation, generators only run when needed. They also save their state between executions.

- The generators have their own local state:

```
(struct generatorV (state-box))
(struct freshG (body env))
(struct suspendedG (thunk))
(struct doneG ())
```

- This function of the yield construct saves a generator's current state…

```
(yk v (lambda () (k v)))
```

- …and this function of the resume construct restores a saved state.

```
(run-thunk thunk)
```

Test Cases

- Example 1
  - A generator that yields three values.

```
{let1 {g {generator
         {yield 10}
         {yield 20}
         {yield 30}}}
     {+ {resume g} {resume g}}}
```

  - Result: 30

- Example 2
  - Using all yielded values.

```
{let1 {g1 {generator
         {yield 1}
         {yield 2}
         {yield 3}}}
     {+ {resume g} {+ resume g}
     {resume g}}}}
```

  - Result: 6

Test Cases cont.

- Example 3
  - Two independent generator instances.

```
{let1 {g1 {generator
        {yield 100}
        {yield 200}}}
    {let1 {g2 {generator
            {yield 1}
            {yield 2}}}
        {+ {resume g1} {resume
        g2}}}})
```

  - Result: 101

- Example 4
  - Interleaved resumes.

```
{let1 {g1 {generator
        {yield 1}
        {yield 2}}}
    {let1 {g2 {generator
            {yield 10}
            {yield 20}}}
        {+ {resume g1}
            {+ {resume g2}
                {+ {resume g1}
                    {resume
                    g2}}}}}}
```

  - Result: 33

Test Cases cont.

- Example 5
  - Generator with computed values (mutation).

```
{let1 {g {generator
         {let1 {n 0}
             {begin
                 {yield n}
                 {set! n {+ n 1}}
                 {yield n}
                 {set! n {+ n 1}}
                 {yield n}}}}}
{+ {resume g} {+ {resume g}
{resume g}}}}
```

  - Result: 3

- Example 6
  - Resuming a finished generator (error).

```
{let1 {g {generator
         {yield 42}}}
    {begin
         {resume g}
         {resume g}}}
```

  - Result: “generator exhausted”

Reflection

A standard function is called by calling its name and any parameters needed, starting from the beginning every single time that it is called and ending upon reaching the end of the function or reaching a return statement. Generators, on the other hand, are called with a resume. The first resume to a generator starts at the beginning. Generators go until reaching a yield statement, returning the value at the yield. Then when resume calls the generator again, the generator starts back after the yield statement and continues until reaching the next yield. This means that the generator's position is saved and can continue at any point, or possibly never continue again if it is not needed. As opposed to returning a function, the state of the generator, all of its non static local variables, and the line it last left of are all preserved. Generators can also be ran without closure, with the generator only having yields or looping a yield

Reflection

Generators are an implementation of lazy evaluation, utilizing it in the form of a function. Allowing long and complicated code to be ran when needed and continue only when needed. While lazy evaluation is typically stateless, generators save their state, or in other words, have their own local stack. In terms of our implementation of generators, the state of the generator is saved and called whenever the generator is resumed making any functions called in the generator apart that state stack and being added and removed from the stack with the generator. If I were to implement generators in a language of my own creation, I would build them off of functions due to their similarities. I'd design them as functions with a global state saved and the yield statements as returns that also save the state of the generator, again due to the similarities between return and yield.